

Optimal Dynamic Sequence Representations *

Gonzalo Navarro[†]

Yakov Nekrich[‡]

Abstract

We describe a data structure that supports access, rank and select operations on a dynamic string $S[1, n]$ over alphabet $[1..\sigma]$ in worst-case time $O(\lg n / \lg \lg n)$, which is optimal. Symbols can be inserted into and deleted from S in $O(\lg n / \lg \lg n)$ amortized time. Those times are better than the best previous dynamic time complexities by a $\Theta(1 + \lg \sigma / \lg \lg n)$ factor. Our structure uses $nH_0(S) + o(n(1 + H_0(S))) + O(\sigma(\lg \sigma + \lg^{1+\varepsilon} n))$ bits, where $H_0(S)$ is the zero-order entropy of S and $0 < \varepsilon < 1$ is any constant. This space redundancy over $nH_0(S)$ is also better, almost always, than that of the best previous dynamic structures, $o(n \lg \sigma) + O(\sigma(\lg \sigma + \lg n))$. We can also handle unbounded alphabets in optimal time, which has been an open problem in dynamic sequence representations.

*Partially funded by Fondecyt grant 1-110066, Chile.

[†]Department of Computer Science, University of Chile. Email: gnavarro@dcc.uchile.cl.

[‡]Department of Computer Science, University of Chile. Email: yakov.nekrich@gmail.com.

1 Introduction

String representations supporting rank and select queries are fundamental in many data structures, including full-text indexes [17, 13, 15], permutations [15, 3], inverted indexes [9, 3], graphs [12], document retrieval indexes [31], labeled trees [15, 5], binary relations [5], and many more. The problem is to encode a string $S[1, n]$ over alphabet $\Sigma = [1..\sigma]$ so as to support the following queries:

$$\begin{aligned} \text{rank}_a(S, i) &= \text{number of occurrences of } a \in \Sigma \text{ in } S[1, i], \text{ for } 1 \leq i \leq n. \\ \text{select}_a(S, i) &= \text{position in } S \text{ of the } i\text{-th occurrence of } a \in \Sigma, \text{ for } 1 \leq i \leq \text{rank}_a(S, n). \\ \text{access}(S, i) &= S[i]. \end{aligned}$$

There exist various representations of S that support these operations [17, 15, 13, 3]. However, these representations are static, that is, S cannot change. In various applications one needs dynamism, that is, to insert and delete symbols in S . There are not many dynamic solutions, however. All are based on the *wavelet tree* representation [17]. The wavelet tree decomposes S hierarchically. In a first level, it separates larger from smaller symbols, by marking in a bitmap which symbols of S were larger and which were smaller. The two subsequences of S are recursively separated. The $\lg \sigma$ levels of bitmaps describe S , and access, rank and select operations on S are carried out via $\lg \sigma$ rank and select operations on the bitmaps. Insertions and deletions in S can also be carried out by inserting and deleting bits from $\lg \sigma$ bitmaps (see Section 2 for more details).

In the static case, rank and select operations on bitmaps take constant time, and therefore access, rank and select on S takes $O(\lg \sigma)$ time [17]. This can be reduced to $O(1 + \lg \sigma / \lg \lg n)$ by using multiary wavelet trees [13]. These separate the symbols into $\rho = o(\lg n)$ ranges, and instead of a bitmap store a sequence over an alphabet of size ρ . In the dynamic case, however, the operations on those bitmaps or sequences are slowed down. Mäkinen and Navarro [22] obtained $O(\lg \sigma \lg n)$ time for all the operations, including updates, by using dynamic bitmaps that handled all the operations in time $O(\lg n)$. They simultaneously compress the sequence to $nH_0(S) + o(n \lg \sigma)$ bits. Here $H_0(S) = \sum_{a \in [1..\sigma]} (n_a/n) \lg(n/n_a) \leq \lg \sigma$ is the zero-order entropy of S , where n_a is the number of occurrences of a in S . González and Navarro [16] improved the times to $O((1 + \lg \sigma / \lg \lg n) \lg n)$ by extending the results to multiary wavelet trees. In this case, instead of dynamic bitmaps, they handled dynamic sequences over a small alphabet (of size ρ). Finally, He and Munro [18] and Navarro and Sadakane [25] obtained the currently best result, $O((1 + \lg \sigma / \lg \lg n) \lg n / \lg \lg n)$ time, still within the same space. They did so by improving the times of the dynamic sequences on small alphabets to $O(\lg n / \lg \lg n)$, which is optimal even on bitmaps [14]. The $\Omega((\lg n / \lg \lg n)^2)$ lower bound for dynamic range counting [28], and the $O(\lg n / \lg \lg n)$ static upper bound using wavelet trees [8], suggest that no more improvements are possible in this line.

In this paper we show that this dead-end can be broken by abandoning the implicit assumption that, to provide access, rank and select on S , we *must* provide rank and select on the bitmaps (or sequences over $[1..\rho]$). We show that all what is needed is to *track* positions of S downwards and upwards along the wavelet tree. It turns out that this tracking can be done in *constant* time per level, breaking the $\Theta(\lg n / \lg \lg n)$ per-level barrier. As a result, we obtain the *optimal* time complexity $O(\lg n / \lg \lg n)$ for the three queries. The update time is also optimal, $O(\lg n / \lg \lg n)$, yet it is amortized. This is $\Theta(\lg \sigma / \lg \lg n)$ times faster than what was believed to be the “ultimate” solution. Moreover, we also improve upon the space by compressing the redundancy of $o(n \lg \sigma)$ of previous dynamic structures. Our space is $nH_0(S) + o(n(1 + H_0(S))) + O(\sigma(\lg \sigma + \lg^{1+\varepsilon} n))$ bits, for any constant $0 < \varepsilon < 1$. Finally, we also handle unbounded alphabets (e.g., $\Sigma = \mathbb{R}$) in optimal time $O(\lg \sigma + \lg n / \lg \lg n)$, a long-standing problem on dynamic sequences. At the end we also describe a number of applications where our result offers new time/space combinations.

2 The Wavelet Tree

Let S be a string over alphabet $\Sigma = [1..\sigma]$. We associate each $a \in \Sigma$ to a leaf v_a of a full balanced binary tree T . The essential idea of the wavelet tree structure is the representation of elements from a string S by bit sequences stored in the nodes of tree T . We associate a subsequence $S(v)$ of S with every node v of T . For the root v_r , $S(v_r) = S$. In general, $S(v)$ consists of all occurrences of symbols $a \in \Sigma_v$ in S , where Σ_v is the set of symbols assigned to leaf descendants of v . The wavelet tree does not store $S(v)$ explicitly, but just a bit vector $B(v)$. We set $B(v)[i] = t$ if the i -th element of $S(v)$ also belongs to $S(v_t)$, where v_t is the t -th child of v (the left child corresponds to $t = 0$ and the right to $t = 1$). This data structure (i.e., T and bit vectors $B(v)$) is called a *wavelet tree*.

For any symbol $S[i] = a$ and every node v such that $a \in \Sigma_v$, there is exactly one bit b_v in $B(v)$ that indicates in which subtree of v the leaf $v_{S[i]}$ is stored. We will say that such b_v *encodes* $S[i]$ in $B(v)$; we will also say that bit b_v from $B(v)$ *corresponds* to a bit b_w from $B(w)$ if both b_v and b_w encode the same symbol $S[i]$. Identifying the positions of bits that encode the same symbol plays a crucial role in wavelet trees. Other, more complex, operations rely on our ability to navigate in the tree and keep track of bits that encode the same symbol.

To implement $\text{access}(S, i)$ we traverse a path from the root to the leaf $v_{S[i]}$. In each visited node we read the bit b_v that encodes $S[i]$ and proceed to the b_v -th child of v . To answer $\text{select}_a(S, i)$, we identify the index of the bit b_{v_r} in $S(v_r)$ that corresponds to $S(v_a)[i]$. To compute $\text{rank}_a(S, i)$, we identify the last bit b' that precedes $B(v_r)[i]$ and corresponds to some symbol in $S(v_a)$.

The standard method used in wavelet trees for identifying the corresponding bits is to maintain rank/select data structures on bit vectors $B(v)$. Suppose that $B(v)[j] = t$; we can find the position of the corresponding bit in the child of v by answering a query $\text{rank}_t(B(v), j)$. If v is the r -th child of a node w , we can find the corresponding bit in w by answering a query $\text{select}_r(B(w), j)$. This approach leads to $O(\lg \sigma)$ query times in the static case because rank/select queries on a bit vector can be answered in constant time. However, we need $\Omega(\lg n / \lg \lg n)$ time to support rank/select and updates on a bit vector [14], which multiplies the operation times. A slight improvement can be achieved by increasing the fan-out of the wavelet tree to $\Theta(\lg^\varepsilon n)$: as before, $B(v)[i] = t$ if the i -th element of $S(v)$ also belongs to $S(v_t)$ for the t -th child v_t of v . This enables us to reduce the height of the wavelet trees and the query time by a $\Theta(\lg \lg n)$ factor. However, it seems that further improvements that are based on dynamic rank/select queries in every node are not possible.

In this paper we use a different approach to identifying the corresponding bits. Instead of storing a bitmap $B(v)$ in an array, we use a compact list structure $L(v)$ to store $B(v)$. Pointers from selected positions in $L(v)$ to the structure $L(w)$ in a parent node w (and vice versa) enable us to navigate between nodes of the wavelet tree in constant time. We extend the idea to multiary wavelet trees. While similar techniques have been used in some geometric data structures [26, 7], applying them on compressed data structures where the bit budget is severely limited is much more challenging and requires new ideas.

3 Basic Structure

We start by describing the main components of our modified wavelet tree. Then, we show how our structure supports $\text{access}(S, i)$ and $\text{select}_a(S, i)$. In the third part of this section we describe additional structures that enable us to answer $\text{rank}_a(S, i)$. Finally, we show how to support updates.

Structure. We assume that the wavelet tree T has node degree $\rho = \Theta(\lg^\varepsilon n)$. We divide sets $B(v)$ into *blocks* and store those blocks in a doubly-linked list $L(v)$. Each block $G_i(v)$, except of the last one, contains $\Theta(\lg^3 n)$ consecutive elements from $B(v)$; the last block contains $O(\lg^3 n)$ consecutive elements. For each $G_i(v)$ we maintain a data structure $R_i(v)$ that supports rank and select queries on elements of $G_i(v)$. Since a block contains a poly-logarithmic number of elements over an alphabet of size ρ , we can answer rank and select queries in $O(1)$ time using $O(|G_i(v)|/\lg^{1-\varepsilon} n)$ additional bits (see Appendix A for details). A *pointer* to an element $B(v)[e]$ consists of two parts: a unique id of the block $G_i(v)$ that contains e and the index of e in $G_i(v)$. We maintain pointers between selected corresponding elements in $L(v)$ and its children. If an element $B(v)[e] = t$ is stored in a block $G_i(u)$ and $B(v)[e'] \neq t$ for any e' in $G_i(u)$ that precedes e , then we store a pointer from e to the corresponding element $B(u_t)[e_t]$ in $L(u_t)$, where u_t is the t -th child of u . If $B(v)[e] = t$ and the corresponding e_t in $L(u_t)$ is the first element in its block, then we also store a pointer from e to e_t . If there is a pointer from e in $L(v)$ to e_t in $L(v_t)$, then we also store a pointer from e_t to e . All these pointers will be called *inter-node pointers*. We describe how inter-node pointers are implemented later in this section.

It is easy to see that the number of inter-node pointers from e in $L(v)$ to e_t in $L(v_t)$ is $\Theta(g(v))$, where $g(v)$ is the number of blocks in $L(v)$. Hence, the total number of pointers that point down from a node v is bounded by $O(g(v)\rho)$. Since this also equals the number of pointers that point up to v , the total number of pointers in the wavelet tree equals $O(\sum_{v \in T} g(v)\rho) = O(n \lg \sigma / \lg^{3-\varepsilon} n + \sigma \lg^\varepsilon n)$. The pointers from a block $G_i(v)$ are stored in a data structure $F_i(v)$. Using $F_i(v)$, we can find, for any element e in $G_i(v)$ and any $1 \leq t \leq \rho$, the last e' preceding e in $G_i(v)$ such that there is a pointer from e' to an element e'_t in $L(v_t)$. We describe in Appendix A how $F_i(v)$ implements the queries in constant time.

For the root node v_r , we store a partial-sum data structure $\mathcal{P}(v_r)$ that contains the number of elements in each block of $L(v_r)$. Using $\mathcal{P}(v_r)$, we can find the block $G_j(v_r)$ that contains the i -th element of $S(v_r) = S$, as well as the number of elements in all blocks that precede a given block $G_j(v_r)$. Both operations can be supported in $O(\lg n / \lg \lg n)$ time [19, 25]. The same data structures $\mathcal{P}(v_a)$ are also stored in the leaves v_a of \mathcal{T} . We observe that we do not store a bitmap $B(v_a)$ in a leaf node v_a . Nevertheless, we divide the (implicit) sequence $B(v_a)$ into blocks and store the number of elements in each block in $\mathcal{P}(v_a)$; we maintain $\mathcal{P}(v_a)$ only if $L(v_a)$ consists of more than one block. Moreover we store inter-node pointers from the parent of v_a to v_a and vice versa. Pointers in a leaf node are constructed according to the same rules as in any other node.

Access and Select Queries. Suppose that the position of an element $B(v)[e] = t$ in $L(v)$ is known, and let i_v be the index of e in its block $G_j(v)$. Then the position of the corresponding e_t in $L(v_t)$ is computed as follows. Using $F_j(v)$, we find the index i' of the largest $e' \leq e$ in $G_j(v)$ such that there is a pointer from e' to e'_t in $L(v_t)$. Due to our construction, such e' must exist. Let i'_v and i'_t denote the indexes of e' and e'_t respectively, and let $G(v_t)$ denote the block that contains e'_t . Let $r_v = \text{rank}_t(G_j(v), i_v)$ and $r'_v = \text{rank}_t(G_j(v), i'_v)$. Due to our rules to define pointers, e_t also belongs to $G(v_t)$ and its index is $i'_t + (r_v - r'_v)$. Thus we can find the position of e_t in $O(1)$ time if the position of $B(v)[e] = t$ is known.

Analogously, assume we know a position $B(v_t)[e_t]$ at $G_j(v_t)$ and want to find the corresponding position e in its parent node v . Using $F_j(v_t)$ we find the last $e'_t \leq e_t$ in $G_j(v_t)$ that has a pointer to its parent, which exists by construction. Let e'_t point to e' , with index i' in a block $G(v)$. Let i'_t and i_t the indexes of e'_t and e_t in $G_j(v_t)$, respectively. Then, by our construction, e is also in $G(v)$ and its index is $\text{select}_t(G(v), \text{rank}_t(G(v), i') + (i_t - i'_t))$.

To solve $\text{access}(S, i)$, we visit nodes $v_0 = v_r, v_1 \dots v_p = v_a$, where v_j is the t_j -th child of v_{j-1} and $B(v_{j-1})[e_{j-1}] = t_j$ encodes $S[i]$. The position of e_0 is found in $O(\lg n / \lg \lg n)$ time using the dynamic partial-sums data structure $\mathcal{P}(v_r)$. If the position of e_{j-1} is known, we can find that of e_j in $O(1)$ time, as explained above. When a leaf node $v_p = v_a$ is reached, we know that $S[i] = a$.

To solve $\text{select}_a(S, i)$, we identify the i -th element in the leaf v_a , using structure $\mathcal{P}(v_a)$. Let e_p be its position. Then we traverse the path $v_p, v_{p-1}, \dots, v_0 = v_r$ where v_{j-1} is the parent of v_j , until the root node is reached. In every node v_j , we find e_{j-1} in $L(v_{j-1})$ that corresponds to e_j as explained above. Finally, we compute the number of elements that precede e_0 in $L(v_r)$ using structure $\mathcal{P}(v_r)$. Clearly, access and select require $O((\lg \sigma + \lg n) / \lg \lg n)$ worst-case time.

Rank Queries. We need some additional data structures for efficient support of rank queries. In every node v such that $L(v)$ consists of more than one block, we store a data structure $P(v)$. Using $P(v)$ we can find, for any $1 \leq t \leq \rho$ and for any block $G_j(v)$, the last block $G_i(v)$ that precedes $G_j(v)$ and contains an element $B(v)[e] = t$. $P(v)$ consists of ρ predecessor data structures $P_t(v)$ for $1 \leq t \leq \rho$. These $P_t(v)$ s could be implemented as van Emde Boas structures, but this would slow down rank to $O(\lg \sigma + \lg n / \lg \lg n)$ time. Instead, we describe in Section 4 a way to support the predecessor queries in constant time.

Suppose that e is the j -th element in a block $G_i(v)$. $P(v)$ enables us to find the position of the last $B(v)[e'] = t$ that precedes e in $B(v)$. First, we use $R_i(v)$ to compute $r = \text{rank}_t(G_i(v), j)$. If $r > 0$, then e' belongs to the same block as e and its index in the block ($G_i(v)$) is $\text{select}_t(G_i(v), r)$. Otherwise, we use $P_t(v)$ to find the last block $G_k(v)$ that precedes $G_i(v)$ and contains an element $B(v)[e'] = t$. We then find the last such element in $G_k(v)$ using $R_k(v)$.

Now we are ready to describe the procedure to answer $\text{rank}_a(S, i)$. The symbol a is represented as a concatenation of symbols $t_0 \circ t_1 \circ \dots \circ t_p$, where each t_j is between 1 and t . We traverse the path from the root $v_r = v_0$ to the leaf $v_a = v_p$. We find the position e_0 of the i -th element S in v_r using the data structure $\mathcal{P}(v_r)$. In each node v_j , $0 \leq j < p$, we identify the position of the last element $B(v_j)[e'_j] = t_j$ that precedes e_j , using $P_{t_j}(v_j)$. Then we find the element e_{j+1} in the list $L(v_{j+1})$ that corresponds to e'_j .

When our procedure reaches the leaf node v_p , the element e_p encodes the last symbol a that precedes $S[i]$. Since we know the position of e_p , we know its index k_1 in its block $G_k(v_p)$. We can find the number k_2 of symbols in all blocks that precede $G_k(v_p)$ using $\mathcal{P}(v_p)$. Clearly, $\text{rank}_a(S, i) = k_1 + k_2$. Because structures P_t answer in time $O(1)$, the overall time for rank is $O((\lg \sigma + \lg n) / \lg \lg n)$.

Updates. Now we describe how inter-node pointers are implemented. We say that an element of $L(u)$ is *pointed* if there is a pointer to this element. Unfortunately, we cannot store the position of a pointed element in the pointer: when a new element is inserted into a block, indexes of all elements that follow it are incremented by 1. Since a block can contain $\Theta(\lg^3 n)$ pointed elements, we would have to update up to $\Theta(\lg^3 n)$ pointers after each insertion and deletion.

Therefore we resort to the following two-level scheme. Each pointed element in a block is assigned a unique id. When a new element is inserted, we assign it the id $\text{max_id} + 1$, where max_id is the maximum id value used so far. We also maintain a data structure $H_i(v)$ for each block $G_i(v)$ that enables us to find the position of a pointed element if its id in $G_i(v)$ is known. Implementation of $H_i(v)$ is based on standard word RAM techniques and a table that contains ids of the pointed elements; details are given in Appendix A.

We describe now how to insert a new symbol a into S at position i . Let e_0, \dots, e_p be the elements that will encode $a = t_0 \circ \dots \circ t_p$. We can find the position of $e_0 = i$ in $L(v_r)$ in $O(\lg n / \lg \lg n)$

time using $\mathcal{P}(v_r)$, and insert t_0 at that position, $B(v_r)[e_0] = t_0$. Now, suppose that the position of $B(v_j)[e_j] = t_j$ in $L(v_j)$ is known. Then we can find the last element $B(v_j)[e'] = t_j$ that precedes e_j in $L(v_r)$; this can be done in the same way as in the procedure for answering rank queries. Once we know the position of e' in $L(v_j)$, we can find the position of e'' in $L(v_{j+1})$ that corresponds to e' . The element t_{j+1} must be inserted into $L(v_{j+1})$ immediately after e'' , at position $e_{j+1} = e'' + 1$.

When a new element $B(v_j)[e_j] = t$ is inserted into $L(v_j)$, we update, if necessary, auxiliary data structures $H_k(v_j)$, $F_k(v_j)$, $R_k(v_j)$, and $P_t(v_j)$, where $G_k(v_j)$ is the block that contains e_j . These updates take $O(1)$ time, see Section 4 for structure $P_t(v_j)$. Since pointers are bidirectional, changes to $F_k(v_j)$ trigger changes in the F structures of v_{j-1} and v_{j+1} . Finally, if v_j is the root node or a leaf, we also update $\mathcal{P}(v_j)$.

If the number of elements in $G_k(v_j)$ exceeds $2\lg^3 n$, we split $G_k(v_j)$ into two blocks, $G_{k_1}(v_j)$ and $G_{k_2}(v_j)$. The sizes of both blocks differ by at most one element. Then, we rebuild the data structures H , F and R for the two new blocks. Note that there are inter-node pointers to $G_k(v_j)$ that now could become dangling pointers, but all those can be known from $F_k(v_j)$, since pointers are bidirectional, and updated to point to the right places in $G_{k_1}(v_j)$ or $G_{k_2}(v_j)$. Finally, if v_j is the root or a leaf, then $\mathcal{P}(v_j)$ is updated.

The total cost of splitting a block is dominated by the cost of building the new data structures H , F and R . We need $O(\lg^3 n)$ time to rebuild them. It is easy to check that we split a block $G_k(v)$ at most once for a sequence of $O(\lg^3 n)$ insertions in $G_k(v)$. Hence, the amortized cost incurred by splitting a block is $O(1)$. Therefore the total cost of an insertion in $L(v)$ is $O(1)$. The insertion of a new symbol leads to $O(\lg \sigma / \lg \lg n)$ insertions into lists $L(v)$. Updates of data structures $\mathcal{P}(v_r)$ and $\mathcal{P}(v_a)$ take $O(\lg n / \lg \lg n)$ time. Hence, the total cost of an insertion is $O((\lg \sigma + \lg n) / \lg \lg n)$. Deletions are handled symmetrically.

Space. We show in Appendix A how to manage the data in blocks $G_i(v)$ so that all the elements stored in lists $L(v)$ use $n \lg \sigma$ bits. Since there are $O(n \lg \sigma / \lg^3 n + \sigma)$ blocks overall, all the pointers between blocks of the same lists add up to $O(n \lg \sigma / \lg^2 n + \sigma \lg n)$ bits. All data structures $\mathcal{P}(v)$ add up to $O(n / \lg^2 n)$ bits. We showed before that the number of inter-node pointers is $O(n \lg \sigma / \lg^{3-\varepsilon} n + \sigma \lg^\varepsilon n)$, hence all inter-node pointers (i.e., F_i and H_i structures) use $O(n \lg \sigma / \lg^{2-\varepsilon} n + \sigma \lg^{1+\varepsilon} n)$ bits. Structures $P_t(v)$ (Section 4) use $O(n \lg \sigma / \lg^{2-\varepsilon} n)$ bits as they have t integers per block. Finally, in Appendix A we show that each structure $R_i(v)$ uses $O(|G_i(v)| / \lg^{1-\varepsilon} n)$ extra bits. Hence, all $R_i(v)$ for all blocks and nodes use $O(n \lg \sigma / \lg^{1-\varepsilon} n)$ bits.

Finally, note that our structures depend on the value of $\lg n$, so they should be rebuilt when $\lceil \lg n \rceil$ changes. Mäkinen and Navarro [22] describe a way to handle this problem without affecting the space nor the time complexities, even in the worst-case scenario. The result is completed in the next section, where we describe the changes needed to implement the predecessor structures.

4 Lazy Deletions and Data Structure $P(u)$

The main idea of our solution is based on lazy deletions: we do not maintain exactly S but a supersequence \overline{S} of it. When a symbol a is deleted from S , we retain it in \overline{S} but take a notice that a is deleted. When the number of deleted symbols exceeds a certain threshold, we expunge from the data structure all the elements marked as deleted. We define $\overline{B}(u)$ and the list $\overline{L}(u)$ for the sequence \overline{S} in the same way as $B(u)$ and $L(u)$ are defined for S .

Since elements of $\overline{L}(u)$ are not deleted, we can implement $P(u)$ as an insertion-only data structure. For any t , $1 \leq t \leq \rho$, we store information about all blocks in a data structure $P_t(u)$. $P_t(u)$

contains one element for each block $G_i(u)$ and is implemented as an incremental split-find data structure that supports insertions in $O(1)$ amortized time and queries in $O(1)$ time [21]. Using $P_t(u)$, we can find for any $G_i(u)$ the last block preceding $G_i(u)$ and containing an element $e = t$. Hence, we can find for any element e in $\bar{L}(u)$ the position of the last element $\bar{B}(u)[e'] = t$ that precedes e . This is done in the same way as in Section 3.

We need some additional data structures to support lazy deletions. A data structure $\bar{\mathcal{P}}(u)$ stores the number of undeleted elements in each block of $\bar{L}(u)$ and supports partial-sum queries. We will maintain $\bar{\mathcal{P}}(u)$ in the root of the wavelet tree and in all leaf nodes. Moreover, we maintain a data structure $D_i(u)$ for every block $G_i(u)$, where u is either the root or a leaf node. $D_i(u)$ can be used to count the number of deleted and undeleted elements before the j -th element in a block $G_i(u)$ for any query index j . The implementation of $D_i(u)$ will be described in Appendix A. We can use $\bar{\mathcal{P}}(u)$ and $D_i(u)$ to find the position τ in $\bar{L}(u)$ where the j -th undeleted element occurs or to count the number of undeleted elements that occur before the position τ in $\bar{L}(u)$.

We also store a list DEL that contains all deleted symbols that have not yet been expunged from the wavelet tree. For any symbol $\bar{S}[i]$ in list DEL we store a pointer to the element e in $\bar{L}(v_r)$ that encodes $\bar{S}[i]$. Pointers in list DEL are implemented in the same way as inter-node pointers.

Queries. Queries are answered very similarly to Section 3. The main idea is that we can essentially ignore deleted elements except at the root and at the leaves.

$\text{access}(S, i)$: Exactly as in Section 3, except that e_0 is the position of the i -th undeleted element in $\bar{L}(v_r)$, found using $\bar{\mathcal{P}}(v_r)$.

$\text{select}_a(S, i)$: We find the position of the i -th undeleted element e_p in $\bar{L}(v_p)$, where $v_p = v_a$, using $\bar{\mathcal{P}}(v_a)$. Then we move up in the tree exactly as in Section 3. When the root node $v_0 = v_r$ is reached, we count the number of undeleted elements that precede e_0 using $\bar{\mathcal{P}}(v_r)$.

$\text{rank}_a(S, i)$: We find the position of the i -th undeleted element e_0 in $\bar{L}(v_0)$ where $v_0 = v_r$. Let t_j be defined as in Section 3. In every node v_j , we find the last element $\bar{B}(v_j)[e'_j] = t_j$ that precedes e_j . Note that this element may be a deleted one, but it still drives us to the correct position in $\bar{L}(v_{j+1})$. We proceed exactly as in Section 3 until we arrive at a leaf $v_p = v_a$. At this point, we count the number of undeleted elements that precede e_p .

Updates. Insertions are supported in the same way as in Section 3. The only difference is that we also update the data structure $D_k(v_j)$ when an element e_j that encodes the inserted symbol a is added to a block $G_k(v_j)$. When a symbol $S[i] = a$ is deleted, we append it to the list DEL of deleted symbols. Then we visit each block $G_k(v_j)$ containing an element e_j that encodes $S[i]$ and update the data structures $D_k(v_j)$. Finally, $\bar{\mathcal{P}}(v_r)$ and $\bar{\mathcal{P}}(v_a)$ are also updated.

When the number of symbols in the list DEL reaches $n/\lg^2 n$, we perform a *cleaning* procedure and get rid of all the deleted elements. Therefore DEL never requires more than $O(n/\lg n)$ bits. For every symbol $\bar{S}[i]$ stored in DEL , we effectively carry out the deletion procedure of Section 3 (recall that the list DEL contains pointers to the positions e_0 in $\bar{L}(v_r)$ to delete). Once all symbols in DEL are processed, we rebuild from scratch the data structures $P(u)$ for all nodes u . The total size of all $P(u)$ structures is $O(n \lg \sigma \lg^\varepsilon n / \lg^3 n)$ elements. Since a data structure for incremental split-find can be constructed in linear time, all $P(u)$ can be rebuilt in $O(n \lg \sigma / \lg^{3-\varepsilon} n)$ time. Hence the amortized time to rebuild the $P(u)$ s is $O(\lg \sigma / \lg^{1-\varepsilon} n)$, which does not affect the amortized time $O((\lg \sigma + \lg n) / \lg \lg n)$ to carry out the effective deletions.

Theorem 1 *A dynamic string $S[1, n]$ over alphabet $[1..\sigma]$ can be stored in a structure using $n \lg \sigma + O(n \lg \sigma / \lg^{1-\varepsilon} n + \sigma \lg^{1+\varepsilon} n)$ bits, for any constant $0 < \varepsilon < 1$, and supporting queries access, rank and select in time $O((\lg \sigma + \lg n) / \lg \lg n)$. Insertions and deletions of symbols are supported in $O((\lg \sigma + \lg n) / \lg \lg n)$ amortized time.*

5 Compressed Space and Optimal Time

We now compress the space of the data structure to zero-order entropy ($nH_0(S)$ plus redundancy, as defined in the Introduction), while improving the time performance to the optimal $O(\lg n / \lg \lg n)$. We first show how a different encoding of the bits within the blocks reduces the $n \lg \sigma$ to $nH_0(S)$ in the space without affecting the time complexities. Then we use this result in combination with alphabet partitioning [3] to obtain the final result, where the redundancy is also compressed.

5.1 Compressing the Bitmaps

Raman et al. [29] describe an encoding for a bitmap $B[1, n]$ that obtains $nH_0(B) + O(n \lg \lg n / \lg n)$ bits of space. It consists of cutting the bitmap into chunks of length $b = \lg(n)/2$ and encoding each chunk i as a pair (c_i, o_i) : c_i is the *class*, which indicates how many 1s are there in the chunk, and o_i is the *offset*, which is the index of this particular chunk within its class. The c_i components add up to $O(n \lg \lg n / \lg n)$ bits, whereas the o_i components add up to $nH_0(B)$. Navarro and Sadakane [25, Sec. 8] describe a technique to maintain a dynamic bitmap in this format. They allow the chunk length b to vary, so they encode triples (b_i, c_i, o_i) as long as the invariant that any pair $b_i + b_{i+1} > b$ is maintained. They show that this retains the space, and that each update affects $O(1)$ chunks.

We extend this encoding to handle an alphabet $[1..\rho]$ [13], so that $b = \lg_\rho(n)/2$ symbols, and each chunk is encoded as a tuple $(b_i, c_i^1, \dots, c_i^\rho, o_i)$ where c_i^t counts the occurrences of t in the block. The “classes” $(b_i, c_i^1, \dots, c_i^\rho)$ use $O(n \lg \lg n / \lg^{1-\varepsilon} n)$ bits, and the offsets still add up to $nH_0(B)$. Blocks are encoded/decoded in $O(1)$ time, as the class part takes $O(\lg^\varepsilon n \lg \lg n) = o(\lg n)$ bits.

In Appendix A we describe how a block is stored as a sequence of miniblocks of $\Theta(\lg n)$ bits, whose length may vary within a constant factor. Those miniblocks, while retaining their “logical” size, will be physically represented using the new encoding, local to each miniblock. As each miniblock will contain a constant number of chunks, it will be processed or updated in constant time. The main difference with respect to the plain representation is that an insertion or deletion may cause the miniblock to grow or shrink by $O(\lg n)$ bits, but we can still handle a constant number of miniblock splits or merges in constant time.

The sum of the local entropies of the chunks, across the whole $L(v)$, adds up to $nH_0(S_v)$, and these add up to $nH_0(S)$ [17]. The redundancy over the entropy is $O(\lg^\varepsilon n \lg \lg n)$ bits per miniblock, adding up to $O(n \lg \sigma \lg \lg n / \lg^{1-\varepsilon} n)$ bits.

Theorem 2 *A dynamic string $S[1, n]$ over alphabet $[1..\sigma]$ can be stored in a structure using $nH_0(S) + O(n \lg \sigma / \lg^{1-\varepsilon} n + \sigma \lg^{1+\varepsilon} n)$ bits, for any constant $0 < \varepsilon < 1$, and supporting queries access, rank and select in time $O((\lg \sigma + \lg n) / \lg \lg n)$. Insertions and deletions of symbols are supported in $O((\lg \sigma + \lg n) / \lg \lg n)$ amortized time.*

5.2 Alphabet Partitioning

The redundancy in Theorem 2 is still a (sublinear) function of $n \lg \sigma$. Now we show how to compress that redundancy as well, while at the same time reducing the time complexities to optimal.

We use a technique inspired in an alphabet partitioning idea [3]. To each symbol a we will assign a level $\ell = \lceil \lg(n/n_a) \rceil$, so that there are at most $\lg n$ levels. Additionally, we assign level $\lceil \lg n \rceil + 1$ to the symbols of Σ not present in S . For each level ℓ we will create a sequence S^ℓ containing the subsequence of S formed by the symbols of level ℓ , with their alphabet remapped to $[1..\sigma_\ell]$, where σ_ℓ is the number of distinct symbols of level ℓ . We will also maintain a sequence of levels S^{lev} , so that $S^{lev}[i]$ is the level of $S[i]$. We represent S^{lev} and the S^ℓ strings using Theorem 2. A few arrays handle the mapping between global symbols of Σ and local symbols in strings S^ℓ : $M[1, \sigma]$ gives the level of each symbol, $N[1, \sigma]$ gives the position of that symbol inside the local alphabet of its level, and local arrays $M^\ell[1, \sigma_\ell]$ map local to global symbols. All these are represented as plain arrays. Thus a symbol $a \in \Sigma$ is represented in string S^ℓ , at level $\ell = M[a]$, where it is written as symbol $a' = N[a]$. Conversely, a symbol a' in S^ℓ corresponds to symbol $a = M^\ell[a'] \in \Sigma$.

Barbay et al. [3] show how operations access, rank, and select on S are carried out via a constant number of operations in S^{lev} and in some S^ℓ . We now extend them to insertions and deletions. To insert symbol a at position i in S , we find its level $\ell = M[a]$ and its translation $a' = N[a]$ inside S^ℓ . Now we insert ℓ at position i in S^{lev} , and a' at position $\text{rank}_\ell(S^{lev}, i)$ in S^ℓ . Deletion is similar: after mapping, we delete the position $S^\ell[\text{rank}_\ell(S^{lev}, i)]$ and then the position $S^{lev}[i]$.

If the symbol a we are inserting did not exist in S , it will be assigned the last level $\ell = \lceil \lg n \rceil + 1$ and will not appear in M^ℓ . In this case we add a at the end of M^ℓ , $M^\ell[\sigma_\ell + 1] = a$, increase σ_ℓ , set $N[a] = \sigma_\ell$ and $M[a] = \ell$. Then we proceed with the insertion. Instead, if a deletion removes the last occurrence of a , we handle it as a part of the more global update mechanism we explain next.

We keep track of the current frequency in S of each symbol $a \in \Sigma$, n_a , and the frequency that a had when it was assigned its current level, n'_a . We retain the level ℓ assigned to a symbol a as long as $n'_a/2 < n_a < 2n'_a$. When $n_a = 2n'_a$ or $n_a = \lfloor n'_a/2 \rfloor$, we move a to a new level $\ell' = \lceil \lg(n/n_a) \rceil = \ell \pm 1$, as follows. We compute the mapping $a' = N[a]$ of a in S^ℓ , change $M[a]$ to ℓ' , and compute the new mapping $a'' = \sigma_{\ell'} + 1$ of a in $S^{\ell'}$. Now, for each of the n_a occurrences of a' in S^ℓ , say $S^\ell[i] = a'$ (found using $i = \text{select}_{a'}(S^\ell, 1)$), we compute its position $j = \text{select}_\ell(S^{lev}, i)$ in S^{lev} , change $S^{lev}[j]$ to ℓ' , remove symbol $S^\ell[i]$, and insert symbol a'' in $S^{\ell'}$ at position $\text{rank}_{\ell'}(S^{lev}, j)$. We also update the mappings: we set $M^{\ell'}[a''] = a$ and $N[a] = a''$, and move the last element of M^ℓ to occupy the empty slot left by a : $M^\ell[a'] = M^\ell[\sigma_\ell]$ and $N[b] = a'$, where $b = M^\ell[\sigma_\ell]$. We find all the occurrences of σ_ℓ in S^ℓ and replace them by a' . Finally, we increase $\sigma_{\ell'}$ and decrease σ_ℓ . Of course when the target of a is the level $\ell' = \lceil \lg n \rceil + 1$, we simply delete it instead of moving it to $S^{\lceil \lg n \rceil + 1}$. Likewise we also remember the total number of symbols n' in the sequence at the time when the data structure was created. When $n = 2n'$ or $n = n'/2$, we rebuild the data structure.

The number of insertions or deletions that must occur until we change the level of a is $n'_a = \Theta(n_a)$. Therefore, the process of changing a symbol from one level to another, which costs $O(n_a)$ update operations on S^{lev} , S^ℓ , M^ℓ , M and N , is amortized over $\Theta(n_a)$ updates. The same occurs with the symbol b mapped to σ_ℓ in S^ℓ , whose occurrences have to be re-encoded as a' : Since $\lceil \lg(n/n'_b) \rceil = \lceil \lg(n/n'_a) \rceil$, it follows that $n_b = \Theta(n_a)$.

Note that we are letting the alphabet of the sequences S^ℓ grow and shrink, which our wavelet trees do not support. Rather, we create them with the maximum possible alphabet size. Since $\ell = \lceil \lg(n/n'_a) \rceil$, the smallest frequency n_a that can belong to level ℓ satisfies $\ell = \lg(n/n'_a)$ and $n_a = n'_a/2$. The total length of the sequence, n , can also grow by a factor of almost 2; therefore

there can be at most $n/n_a = 2^{\ell+2}$ distinct symbols at level ℓ .

Time. The queries on S^{lev} cost time $O(\lg n / \lg \lg n)$, because its alphabet is of size $O(\lg n)$. Queries on S^ℓ cost time $O((\lg 2^\ell + \lg n) / \lg \lg n) = O(\lg n / \lg \lg n)$, since $\ell = O(\lg n)$. The accesses to M , N , and M^ℓ are constant-time. Therefore, we reach the optimal worst-case time $O(\lg n / \lg \lg n)$ for the three queries. Updates, similarly, cost $O(\lg n / \lg \lg n)$ amortized time.

Space. Each symbol a with frequency n_a will be stored at a level $\ell = \lceil \lg(n/n_a) \rceil \pm 2$, which is a sequence over an alphabet of size $2^{\ell+O(1)}$. Therefore, calling $n_\ell = |S^\ell|$, we will spend $n_a \lg(n_\ell/n_a) + O(n_a \lg(2^{\ell+2})/\lg^{1-\varepsilon} n)$ bits for it, according to Theorem 2. This is $n_a \lg(n_\ell/n_a) + o(n_a \lg(n/n_a))$ bits, which added over the whole S^ℓ yields $\sum n_a \lg(n_\ell/n_a) + o(\sum n_a \lg(n/n_a))$ bits. Now consider the occurrences of symbol ℓ in S^{lev} , which we will also charge to S^ℓ . These cost $n_\ell \lg(n/n_\ell) + O(n_\ell \lg \lg n / \lg^{1-\varepsilon} n) = n_\ell \lg(n/n_\ell) + o(n_\ell)$. Added to the space spent at S^ℓ itself, and since the sum of the n_a 's is n_ℓ , we obtain $\sum n_a \lg(n/n_a) + o(\sum n_a (1 + \lg(n/n_a)))$ bits. Now, adding over the symbols a of all the levels, we obtain the total space $nH_0(S) + o(n(1 + H_0(S)))$. Theorem 2 also involves a cost of $O(2^{\ell+2} \lg^{1+\varepsilon} n)$ bits per level ℓ , which seem to add up to $O(n \lg^{1+\varepsilon} n)$. However, this blowup is artificial since it comes from storing at most one almost-empty block at each wavelet tree node, and there cannot be more than σ nonempty wavelet tree nodes across all the strings S^ℓ . Thus the extra space is only $O(\sigma \lg^{1+\varepsilon} n)$.

In addition we spend $O(\sigma(\lg n + \lg \sigma))$ bits for the arrays M , N and M^ℓ . Finally, recall that we also spend space in storing deleted symbols, but these are at most $O(n/\lg^2 n)$, and thus they cannot increase the entropy by more than $O(n/\lg n)$. We have obtained the final result.

Theorem 3 *A dynamic string $S[1, n]$ over alphabet $[1..\sigma]$ can be stored in a structure using $nH_0(S) + O(n(1 + H_0(S))/\lg^{1-\varepsilon} n + \sigma(\lg \sigma + \lg^{1+\varepsilon} n))$ bits, for any constant $0 < \varepsilon < 1$, and supporting queries access, rank and select in optimal time $O(\lg n / \lg \lg n)$. Insertions and deletions of symbols are supported in $O(\lg n / \lg \lg n)$ amortized time.*

5.3 Handling General Alphabets

Our time results do not depend on the alphabet size σ , yet our space does, in a way that ensures that σ gives no problems as long as $\sigma = o(n/\lg^{1+\varepsilon} n)$ for some constant $\varepsilon > 0$.

Let us now consider the case where the alphabet Σ is much larger than the *effective* alphabet of the string, that is, the set of symbols that actually appear in S at a given point in time. Let us now use $\sigma \leq n$ to denote the effective alphabet size. Our aim is to maintain the space within $nH_0(S) + O(n(1 + H_0(S))/\lg^{1-\varepsilon} n + \sigma \lg^{1+\varepsilon} n)$ bits, even when the symbols come from a large universe $\Sigma = [1..|\Sigma|]$, or even from an unbounded ordered universe such as $\Sigma = \mathbb{R}$ or $\Sigma = \Gamma^*$ (i.e., Σ are words over strings over another alphabet Γ).

Our arrangement into strings S^ℓ gives a simple way to handle a sequence over an unbounded ordered alphabet. By changing these tables to binary search trees and maintaining the alphabets of strings S^ℓ at size $O(\min(\sigma, 2^\ell))$, we retain the space in terms of σ , and the times raise to $O(\tau \lg \sigma + \lg n / \lg \lg n)$, where τ is the time of a single comparison between elements of Σ (e.g., $\tau = O(1)$ for $\Sigma = \mathbb{R}$ in the comparison model; it is also possible to obtain $O(|w|)$ instead of $O(\tau \lg \sigma)$ on strings w by using tries). This time is optimal if we can only make binary comparisons on Σ . The space redundancy of our data structure increases only by what is needed to represent σ elements from Σ . If, in particular, Σ is an integer range $[1..|\Sigma|]$, the time can be reduced to $O(\lg \lg |\Sigma| + \lg n / \lg \lg n)$ and the space increases by $O(\sigma \lg |\Sigma|)$ bits, by using y-fast tries [32].

6 Applications

Our new results impact in a number of applications that build on dynamic sequences. For lack of space we describe the most obvious one here and defer the rest to Appendix B.

Dynamic sequence collections. The standard application of dynamic sequences, stressed out in several previous papers [11, 22, 16, 25], is to maintain a collection \mathcal{C} of texts, where one can carry out indexed pattern matching, as well as inserting and deleting texts from the collection. With our new representation we can improve the time of previous work (yet our update time is amortized).

Theorem 4 *There exists a data structure for handling a collection \mathcal{C} of texts over an alphabet $[1, \sigma]$ within size $nH_h(\mathcal{C}) + o(n \lg \sigma) + O(\sigma \lg^{1+\epsilon} n + \sigma^{h+1} \lg n + m \lg n + w)$ bits, for any constant $\epsilon > 0$ and simultaneously for all h . Here n is the length of the concatenation of m texts, $\mathcal{C} = 0 T_1 0 T_2 \dots 0 T_m$, and we assume that $\sigma = o(n)$ is the alphabet size and $w = \Omega(\lg n)$ is the machine word size under the RAM model. The structure supports counting of the occurrences of a pattern P in $O(|P| \lg n / \lg \lg n)$ time, and inserting and deleting a text T in $O(\lg n + |T| \lg n / \lg \lg n)$ amortized time. After counting, any occurrence can be located in time $O(\lg n \lg_\sigma n)$. Any substring of length ℓ from any T in the collection can be displayed in time $O(\lg n (\lg_\sigma n + \ell / \lg \lg n))$. For $0 \leq h \leq (\alpha \lg_\sigma n) - 1$, for any constant $0 < \alpha < 1$, the space complexity simplifies to $nH_h(\mathcal{C}) + o(n \lg \sigma) + O(m \lg n + w)$ bits.*

The theorem refers to $H_h(\mathcal{C})$, the h -th order empirical entropy of sequence \mathcal{C} [23]. This is a lower bound to any semistatic statistical compressor that encodes each symbol as a function of the h preceding symbols in the sequence, and it holds $H_h(\mathcal{C}) \leq H_{h-1}(\mathcal{C}) \leq H_0(\mathcal{C}) \leq \lg \sigma$ for any $h > 0$.

To obtain H_h from H_0 we use Theorem 2, which does not use alphabet partitioning (its times become $O(\lg n / \lg \lg n)$ since we assume $\sigma = o(n)$). In that version of our structure the sequence is encoded using tuples, and the space is the sum of the local entropy of the miniblocks. To offer search capabilities, the Burrows-Wheeler Transform (BWT) [10] of \mathcal{C} , \mathcal{C}^{bwt} , is represented, not \mathcal{C} . It has been proved [22] that if one splits the sequences stored at the levels of the wavelet tree of \mathcal{C}^{bwt} into chunks representing $\Theta(\lg n)$ bits, then the sum of the zero-order entropies of the chunks add up to $nH_h(\mathcal{C}) + O(\sigma^{h+1} \lg n)$ bits, and this stays true for the variable- b chunks we use [25]. Finally, the locating and displaying overheads are obtained by marking one element out of $\lg_\sigma n \lg \lg n$.

7 Conclusions and Further Challenges

We have obtained $O(\lg n / \lg \lg n)$ time for all the operations that handle a dynamic sequence on an arbitrary (known) alphabet $[1.. \sigma]$, matching lower bounds that apply to binary alphabets [14]. Our structure is faster than previous work [18, 25] by a factor of $\Theta(\lg \sigma / \lg \lg n)$. It also compresses the redundancy space, using $nH_0(S) + o(n(1 + H_0(S))) + O(\sigma(\lg \sigma + \lg^{1+\epsilon} n))$ bits, instead of $nH_0(S) + o(n \lg \sigma) + O(\sigma(\lg \sigma + \lg n))$ of previous work. We can also handle unbounded alphabets. Our result can be applied to a number of problems; we have described several ones.

The only remaining advantage of previous work [18, 25] is that their update times are worst-case, whereas in our structure they are amortized. Obtaining optimal worst-case time complexity for updates is an interesting future challenge.

Another challenge is to simulate other operations than access, rank and select. Obtaining the full functionality of wavelet trees with better time than the current dynamic ones [18, 25] is unlikely, as then we could probably solve range counting on an $n \times n$ grid [8] in less than the dynamic lower bound, $\Theta((\lg n / \lg \lg n)^2)$ [28]. Yet, there may be some intermediate functionality of interest.

References

- [1] R. Baeza-Yates and B. Ribeiro. *Modern Information Retrieval*. Addison-Wesley, 2nd edition, 2011.
- [2] J. Barbay, F. Claude, and G. Navarro. Compact rich-functional binary relation representations. In *Proc. LATIN*, pages 170–183, 2010.
- [3] J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet partitioning for compressed rank/select and applications. In *Proc. ISAAC*, pages 315–326 (II), 2010.
- [4] J. Barbay, A. Golynski, I. Munro, and S. S. Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. *Theor. Comp. Sci.*, 387(3):284–297, 2007.
- [5] J. Barbay, M. He, I. Munro, and S. S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. SODA*, pages 680–689, 2007.
- [6] J. Barbay and G. Navarro. Compressed representations of permutations, and applications. In *Proc. STACS*, pages 111–122, 2009.
- [7] G. E. Blelloch. Space-efficient dynamic orthogonal point location, segment intersection, and range reporting. In *Proc. SODA*, pages 894–903, 2008.
- [8] P. Bose, M. He, A. Maheshwari, and P. Morin. Succinct orthogonal range search structures on a grid with applications to text indexing. In *Proc. WADS*, pages 98–109, 2009.
- [9] N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *Proc. SIGIR*, pages 139–146, 2008.
- [10] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
- [11] H. Chan, W. Hon, T. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Trans. Alg.*, 3(2):21, 2007.
- [12] F. Claude and G. Navarro. Extended compact Web graph representations. In *Algorithms and Applications (Ukkonen Festschrift)*, pages 77–91. Springer, 2010.
- [13] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Alg.*, 3(2):article 20, 2007.
- [14] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proc. STOC*, pages 345–354, 1989.
- [15] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. SODA*, pages 368–373, 2006.
- [16] R. González and G. Navarro. Rank/select on dynamic compressed sequences and applications. *Theor. Comp. Sci.*, 410:4414–4422, 2009.
- [17] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850, 2003.

- [18] M. He and I. Munro. Succinct representations of dynamic strings. In *Proc. SPIRE*, pages 334–346, 2010.
- [19] W.-K. Hon, K. Sadakane, and W.-K. Sung. Succinct data structures for searchable partial sums. In *Proc. ISAAC*, pages 505–516, 2003.
- [20] W. K. Hon, K. Sadakane, and W. K. Sung. Breaking a Time-and-Space Barrier in Constructing Full-Text Indices. *SIAM J. Comp.*, 38(6):2162–2178, 2009.
- [21] H. Imai and T. Asano. Dynamic segment intersection search with applications. In *Proc. FOCS*, pages 393–402, 1984.
- [22] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 4(3):article 32, 2008.
- [23] G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.
- [24] J. I. Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log n)$ time. *J. Comp. Sys. Sci.*, 33(1):66–74, 1986.
- [25] G. Navarro and K. Sadakane. Fully-functional static and dynamic succinct trees. *CoRR*, abs/0905.0768v5, 2010.
- [26] Y. Nekrich. A dynamic stabbing-max data structure with sub-logarithmic query time. In *Proc. ISAAC*, pages 170–179, 2011.
- [27] D. Okanohara and K. Sadakane. A linear-time Burrows-Wheeler transform using induced sorting. In *Proc. SPIRE*, LNCS 5721, pages 90–101, 2009.
- [28] M. Patrascu. Lower bounds for 2-dimensional range counting. In *Proc. STOC*, pages 40–46, 2007.
- [29] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k -ary trees and multisets. In *Proc. SODA*, pages 233–242, 2002.
- [30] R. Raman and S. S. Rao. Succinct dynamic dictionaries and trees. In *Proc. ICALP*, pages 357–368, 2003.
- [31] N. Välimäki and V. Mäkinen. Space-efficient algorithms for document retrieval. In *Proc. CPM*, pages 205–215, 2007.
- [32] D. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Inf. Proc. Lett.*, 17(2):81–84, 1983.

A Data Structures for Handling Blocks

We describe the way the data is stored in blocks $G_i(u)$, as well as the way the various data structures inside blocks operate. All data structures are based on the same idea: We maintain a tree with node degree $\lg^\delta n$ and leaves that contain $o(\lg n)$ elements. Since elements within a block can be addressed with $O(\lg \lg n)$ bits, each internal node and each leaf fits into one machine word. Moreover, we can support searching and basic operations in each node in constant time.

Data organization. The block data is physically stored as a sequence of *miniblocks* of $\Theta(\lg_\rho n)$ symbols. Thus there are $O(\lg^2 n \lg \rho) = O(\lg^2 n \lg \lg n)$ miniblocks in a block. These miniblocks will be the leaves of a τ -ary tree T , for $\tau = \Theta(\lg^\delta n)$ and some constant $0 < \delta < 1$. The height of this tree is constant, $O(1/\delta)$. Each node of T stores τ counters telling the number of symbols stored at the leaves that descend from each child. This requires just $O(\tau \lg \lg n) = o(\lg n)$ bits. To access any position of $G_i(u)$, we descend in T , using the counters to determine the correct child. When we arrive at the leaf, we know the local offset of the desired symbol within the leaf, and can access it directly. Since the counters fit in less than a machine word, a small universal table gives the correct child in constant time, therefore we have $O(1)$ access to any symbol (actually to any $\Theta(\lg_\rho n)$ consecutive symbols).

Upon insertions or deletions, we arrive at the correct leaf, insert or delete the symbol (in constant time because the leaf contains $\Theta(\lg n)$ bits overall), and update the counters in the path from the root (in constant time as they have $o(\lg n)$ bits). The leaves may have $\lg n$ to $2 \lg n$ bits. Splits/merges upon overflows/underflows are handled as usual, and can be solved in a constant number of operations requiring $O(1)$ time each (T operates as a B-tree; internal nodes may have τ to 2τ children).

The space overhead due to the nodes of T is $O(|G_i(u)| \lg^\delta n \lg \lg n / \lg n)$ bits. We consider now the space used by the data itself.

In order not to waste space, the miniblock leaves are stored using a memory management technique by Munro [24]. For our case, it allows us to allocate, free, and access miniblocks of length $\lg n$ to $2 \lg n$ in constant time. Its space waste, given that our pointers are of $O(\lg \lg n)$ bits, is $O(\lg \lg n / \lg n)$ per allocated miniblock, plus a global redundancy of $O(\lg^2 n)$ bits.¹ We use one structure per block, handling its miniblocks, so the global redundancy adds just $O(n / \lg n)$ bits overall. Each structure uses a memory area of fixed-size cells (inside which the variable-length miniblocks are stored) that grows or shrinks at the end as miniblocks are created or destroyed. A structure giving that functionality is called an *extendible array (EA)* [30]. We need to handle a set of $O(n / \lg^3 n)$ EAs, what is called a *collection of extendible arrays*. Its functionality includes accessing any cell of any EA, letting it grow or shrink by one cell, and create and destroy EAs. The following lemma, simplified from the original [30, Lemma 1], and using words of $\lg n$ bits, is useful.

Lemma 1 *A collection of a EAs of total size s bits can be represented using $s + O(a \lg n + \sqrt{sa \lg n})$ bits of space, so that the operations of creation of an empty EA and access take constant worst-case time, whereas grow/shrink take constant amortized time. An EA of s' bits can be destroyed in time $O(s' / \lg n)$.*

¹When we store the miniblocks in compressed form, in Section 5.1, their physical size could be as small as $O(\lg^\varepsilon n \lg \lg n)$. Such small sizes can also be handled, and the overhead with respect to the amount of logical data represented is maintained.

In our case $a = O(n/\lg^3 n)$ and $s = O(n \lg \sigma)$, so the extra space used is $O(n/\lg^2 n + n\sqrt{\lg \sigma}/\lg n) = o(n \lg \sigma/\lg n)$.

Structure $R_i(u)$: To support rank and select we enrich T with further information per node. We store ρ counters with the number of occurrences of each symbol in the subtree of each child. The node size becomes $O(\tau \rho \lg \lg n) = O(\lg^{\varepsilon+\delta} n \lg \lg n) = o(\lg n)$ as long as $\varepsilon + \delta < 1$. This dominates the total space overhead, which becomes $O(n \lg \sigma \lg \lg n / \lg^{1-\varepsilon-\delta} n)$. By setting δ and ε to slightly more than the original ε value, this can be rewritten as $O(n \lg \sigma / \lg^{1-\varepsilon} n)$.

With this information on the nodes we can easily solve rank and select in constant time, by descending on T and determining the correct child (and accumulating data on the leftward children) in $O(1)$ time using universal tables. Nodes can also be updated in constant time even upon splits and merges, since all the counters can be recomputed in $O(1)$ time.

Structure $F_i(u)$. This structure stores all the intra-node pointers leaving from block $G_i(u)$, to its parent and to any of the ρ children of the wavelet tree node.

The structure is a tree T_f very similar in structure to T . The pointers stored are intra-node, and thus require $\Theta(\lg n)$ bits. Thus we store a constant number of pointers per leaf. For each pointer we store the position in $G_i(u)$ holding the pointer (relative to the starting position of the leaf node) and the target position. The internal nodes, of arity τ , maintain information on the number of positions of $G_i(u)$ covered by each child, and the number of pointers of each kind ($1 + \rho$ counters) stored in the subtree of each child. This requires $O(\tau \rho \lg \lg n) = o(\lg n)$ bits, as before. To find the last position before j holding a pointer of a certain kind (parent or t -th wavelet tree child, for any $1 \leq t \leq \rho$), we traverse T_f from the root looking for position j . At each node v , it might be that the child u where we have to enter holds pointers of that kind, or not. If it does, then we first enter into child u . If we return with an answer, we recursively return it. If we return with no answer, or there are no pointers of the desired kind below u , we enter into the last sibling to the left of u that holds a pointer of the desired kind, and switch to a different mode where we simply go down the tree looking for the rightmost child with a pointer of the desired kind. It is not hard to see that this procedure visits $O(1/\delta)$ nodes, and thus it is constant-time because all the computations inside nodes can be done in $O(1)$ time with universal tables. When we arrive at the leaf, there may be at most two pointers associated to the desired position (one to the parent and another to a wavelet tree child), so we can scan for the desired pointer in constant time.

The tree T_f is updated upon insertions and deletions of pointers just like T . It must also be updated upon insertions and deletions of symbols, even if they do not have pointers. We traverse T_f looking for the position of the update, change the offsets stored at the leaf, and update the subtree sizes stored at the nodes.

Structure $H_i(u)$. This structure manages the inter-node pointers that point inside $G_i(u)$. As explained in the paper, we give a handle to the outside nodes, that do not change over time, and $H_i(u)$ translates handles to positions in $G_i(u)$.

We store a tree T_h that is just like T_f , where the incoming pointers are stored. T_h is simpler, however, because at each node we only need to store the number of positions covered by the subtree of each child. Also, it is possible to traverse T_h from a leaf to the root. We also manage a table Tbl so that $Tbl[h]$ points to the leaf where the pointer corresponding to handle h is stored. At the leaves we store, for each pointer, a backpointer to Tbl and the position in $G_i(u)$ (in relative form). Given a handle h , we go to the leaf, find in constant time the one pointing back to h , and move

upwards up to the root, adding to the position the number of positions covered by leftward children of each node. At the end we have obtained the position in constant time.

When pointers to $G_i(u)$ are created or destroyed, we insert or remove pointers in T_h . We maintain a list of empty cells in Tbl for future handles. We must also update T_f upon symbol insertions and deletions in $G_i(u)$, to maintain the positions up to date. When a leaf splits or merges, we update the pointers from a constant number of positions in Tbl , found with the backpointers. Since Tbl contains $O(\lg^3 n)$ pointers of $O(\lg \lg n)$ bits, it poses an overhead of $O(\lg \sigma \lg \lg n / \lg n)$ per symbol, so we can allocate it for the maximum possible block size.

Structure $D_i(u)$. This is a simple tree T_d similar to T , storing at each node the number of positions and the number of undeleted positions below each child. It should be obvious at this point how to operate it.

B More Applications

Burrows-Wheeler Transform in compressed space. Another application of dynamic sequences is to build the BWT of a text T , T^{bwt} , within compressed space, by starting from an empty sequence and inserting each new character, $T[n]$, $T[n-1]$, \dots , $T[1]$, at the proper positions. The result is stated as the compressed construction of a static FM-index [13], a compressed index that consists essentially on a (static) wavelet tree of T^{bwt} . Our new representation improves upon the best previous result on compressed space [25].

Theorem 5 *The Alphabet-Friendly FM-index [13], as well as the BWT [10], of a text $T[1, n]$ over an alphabet of size σ , can be built using $nH_h(T) + o(n \lg \sigma)$ bits, simultaneously for all $1 \leq h \leq (\alpha \lg_\sigma n) - 1$ and any constant $0 < \alpha < 1$, in time $O(n \lg n / \lg \lg n)$. It can also be built within the same time and $nH_0(T) + o(n(1 + H_0(T))) + O(\sigma(\lg \sigma + \lg^{1+\epsilon} n))$ bits, for any constant $\epsilon > 0$ and any alphabet size σ .*

We are using Theorem 2 for the case $h > 0$, and Theorem 3 to obtain a less alphabet-restrictive result for $h = 0$. Note we obtain $o(n \lg n)$ time within compressed space. Other space-conscious results that achieve better time complexity (but more space) than our result are Okanohara and Sadakane [27], who achieved optimal $O(n)$ time within $O(n \lg \sigma \lg \lg_\sigma n)$ bits of space, and Hon et al. [20], who achieved $O(n \lg \lg \sigma)$ time and $O(n \lg \sigma)$ bits of space.

Binary relations. Barbay et al. [4] show to represent a binary relation of t pairs relating n “objects” with σ “labels” by means of a string of t symbols over alphabet $[1..\sigma]$ plus a bitmap of length $t + n$. The idea is to traverse the matrix, say, object-wise, and write down in a string the labels of the pairs found. Meanwhile we append a 1 to the bitmap each time we find a pair and a 0 each time we move to the next object. Then queries like: find the objects related to a label, find the labels related to an object, and tell whether an object and a label are related, are answered via access, rank and select operations on the string and the bitmap.

A limitation in the past to make this representation dynamic was that creating or removing labels implied changing the alphabet of the string. Now we can use Theorem 3 and the results of Section 5.3 to obtain a fully dynamic representation. Note our structure is so efficient that the bitmap times dominate the complexity.

Theorem 6 *A dynamic binary relation consisting of t pairs relating n objects with σ labels can support the operations of counting and listing the objects related to a given label, counting and listing*

the labels related to a given object, and telling whether an object and a label are related, all in time $O(\lg(n+t)/\lg \lg(n+t))$ per delivered datum. Pairs can also be added and deleted in amortized time $O(\lg(n+t)/\lg \lg(n+t))$. The space required is $tH + n + t + o(n+t+tH) + O(\sigma(\lg \sigma + \lg^{1+\epsilon} t))$, where ϵ is any constant and $H = \sum_{1 \leq i \leq \sigma} (t_i/t) \lg(t/t_i) \leq \lg \sigma$, where t_i is the number of objects related to label i . It is also possible to handle insertions and deletions of labels and objects (deletions need that no pair involves them), coming from a universe $[1..N]$, within $O(\lg \lg N)$ additional time per operation and $O((n + \sigma) \lg N)$ extra bits of space.

Directed graphs. A particularly interesting and general binary relation is a directed graph with n nodes and e edges. Our binary relation representation allows one to navigate it in forward and backward direction, and modify it, within little space.

Theorem 7 *A dynamic directed graph consisting of n nodes and e edges can support the operations of counting and listing the neighbors pointed from a node, counting and listing the reverse neighbors pointing to a node, and telling whether there is a link from one node to another, all in time $O(\lg(n+e)/\lg \lg(n+e))$ per delivered datum. Edges can also be added and deleted in amortized time $O(\lg(n+e)/\lg \lg(n+e))$. The space required is $eH + n + e + o(n+e+eH) + O(n(\lg n + \lg^{1+\epsilon} e))$, where ϵ is any constant and $H = \sum_{1 \leq i \leq \sigma} (e_i/e) \lg(e/e_i) \leq \lg n$, where e_i is the outdegree of node i . It is also possible to handle insertions and deletions of nodes (deletions need that the node is disconnected from the graph), coming from a universe $[1..N]$, within $O(\lg \lg N)$ additional time per operation and $O(n \lg N)$ extra bits of space.*

Note that we can change “outdegree” by “indegree” in the theorem by representing the transposed graph, as operations are symmetric. We can similarly transpose general binary relations.

Inverted indexes. Finally, we consider an application where the symbols are words. Take a text T as a sequence of n words, which are strings over a set of letters Γ . The alphabet of T is $\Sigma = \Gamma^*$, and its effective alphabet is called the *vocabulary* V of T , of size $|V| = \sigma$. A *positional inverted index* is a data structure that, given a word $w \in V$, tells the positions in T where w appears [1].

A well known way to simulate a positional inverted index within no extra space on top of the compressed text is to use a compressed sequence representation for T (over alphabet Σ), so that operation $\text{select}_w(T, i)$ simulates access to the i th position of the list of word w , whereas access to the original T is provided via $\text{access}(T, i)$. Operation rank can be used to emulate various inverted index algorithms, particularly for intersections [6]. The space is the zero-order entropy of the text seen as a sequence of words, which is very competitive in practice. Our new technique permits modifying the underlying text, that is, it simulates a dynamic inverted index. For this sake we use the technique of Section 5.3, and a trie to handle the vocabulary.

Theorem 8 *A text of n words with a vocabulary of size σ can be represented within $nH_0(T) + o(n(1 + H_0(T))) + O(\sigma(\lg \sigma + \lg^{1+\epsilon} n))$ bits of space, where $\epsilon > 0$ is an arbitrary constant and $H_0(T)$ is the word-wise entropy of T . The representation outputs any word $T[i] = w$ given i , finds the position of the i th occurrence of any word w , and tells the number of occurrences of any word w up to position i , all in time $O(|w| + \lg n / \lg \lg n)$. A word w can be inserted or deleted at any position in T in amortized time $O(|w| + \lg n / \lg \lg n)$.*

Another kind of inverted index, a *non-positional* one, relates each word with the documents where it appears (not to the exact positions). This can be seen as a direct application of our binary relation representation [2], and our dynamization theorems apply to it as well.